LA-UR-22-20788

Approved for public release; distribution is unlimited.

Title:	Fast BLT Code
Author(s):	Nelson, Eric Michael
Intended for:	Report
Issued:	2022-01-31

Los Alamos
NATIONAL LABORATORY

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Fast BLT Code

Eric M Nelson

27 January 2022

This note discusses numerical algorithmic software design considerations and performance estimates for a fast BLT coupling code [1] written in c++. The aim of this code is to conduct faster parameter studies over line orientations. The original matlab code was written by Mike Rivera. Art Barnes ported this code to julia. I rewrote portions of the code for speed improvement, mainly to eliminate some redundant computation when calculating many line orientations. But this code is still far from optimal.

Most of the compute time will ultimately be in inverse FFTs. Most of the work outside of the inverse FFTs is calculating the inputs to the inverse FFTs: the fourier transforms of the coupled terminal voltages and currents. This note is mostly focused on calculating these inverse FFT inputs.

Common Terms

Many terms in the BLT formulation are independent of the line orientation ϕ, and thus do not need to be recalculated for each line orientation. This section identifies the common terms that do not need to be recalculated for each line orientation, and how those terms are used in the remaining calculations for each line orientation.

Let $\tilde{E}_\theta = \tilde{E}_0 \cos\alpha$ be the fourier transform of the vertical (or polar) electric field component E_θ, and let $\tilde{E}_\phi = -\tilde{E}_0 \sin\alpha$ be the fourier transform of the horizontal (or azimuthal) electric field component E_ϕ. The minus sign is due to the coordinate system employed by the BLT formulation as described by Tesche differing from the high-altitude EMP codes.

The first source term is

$$S_1 = \frac{e^{\gamma L}}{2}(e^{-\chi L} - e^{-\gamma L})\left(\frac{\tilde{E}_\theta \sin\psi\cos\phi\ W_\theta - \tilde{E}_\phi \sin\phi\ W_\phi}{\gamma - \chi} + j\frac{\tilde{E}_\theta \cos\psi}{k\sin\psi}(W_\theta - 1 + R_v)\right)$$

where $\chi = jk\cos\psi\cos\phi$ and

$$W_\theta = e^{jkh\sin\psi} - R_v e^{-jkh\sin\psi}$$
$$W_\phi = e^{jkh\sin\psi} + R_h e^{-jkh\sin\psi}.$$

The second source term is

$$S_2 = \frac{e^{\gamma L}}{2}(e^{-\chi L}e^{-\gamma L} - 1)\left(\frac{\tilde{E}_\theta \sin\psi\cos\phi\ W_\theta - \tilde{E}_\phi \sin\phi\ W_\phi}{\gamma + \chi} - j\frac{\tilde{E}_\theta \cos\psi}{k\sin\psi}(W_\theta - 1 + R_v)\right).$$

We will calculate $S_1' = S_1 e^{-\gamma L}$ and $S_2' = S_2 e^{-\gamma L}$ in order to avoid overflow. The five complex terms or factors independent of ϕ that we will precalculate are:

$$-\gamma L, \qquad e^{-\gamma L}, \qquad \tilde{E}_{x\theta} = \tilde{E}_\theta W_\theta, \qquad \tilde{E}_{x\phi} = \tilde{E}_\phi W_\phi, \qquad \tilde{V}_t = j\frac{\tilde{E}_\theta \cos\psi}{2k\sin\psi}(W_\theta - 1 + R_v).$$

Then

$$S_1' = (e^{-\chi L} - e^{-\gamma L})\left(\frac{\tilde{E}_{x\theta}\sin\psi\cos\phi - \tilde{E}_{x\phi}\sin\phi}{\gamma L - \chi L}\frac{L}{2} + \widetilde{V}_t\right)$$
$$S_2' = (1 - e^{-\chi L}e^{-\gamma L})\left(\frac{\tilde{E}_{x\theta}\sin\psi\cos\phi - \tilde{E}_{x\phi}\sin\phi}{-\gamma L - \chi L}\frac{L}{2} + \widetilde{V}_t\right).$$

Once the source terms are known, voltages and currents at the line terminations are calculated via the matrix equations

$$\begin{bmatrix} V(0) \\ V(L) \end{bmatrix} = \begin{bmatrix} 1+\rho_1 & 0 \\ 0 & 1+\rho_2 \end{bmatrix}\begin{bmatrix} -\rho_1 e^{-\gamma L} & 1 \\ 1 & -\rho_2 e^{-\gamma L} \end{bmatrix}^{-1}\begin{bmatrix} S_1' \\ S_2' \end{bmatrix} = \mathrm{A_V}\begin{bmatrix} S_1' \\ S_2' \end{bmatrix}$$

and

$$\begin{bmatrix} I(0) \\ I(L) \end{bmatrix} = \frac{1}{Z_c}\begin{bmatrix} 1-\rho_1 & 0 \\ 0 & 1-\rho_2 \end{bmatrix}\begin{bmatrix} -\rho_1 e^{-\gamma L} & 1 \\ 1 & -\rho_2 e^{-\gamma L} \end{bmatrix}^{-1}\begin{bmatrix} S_1' \\ S_2' \end{bmatrix} = \mathrm{A_I}\begin{bmatrix} S_1' \\ S_2' \end{bmatrix}$$

where ρ_1 and ρ_2 are the voltage reflection coefficients at the two ends $x = 0$ and $x = L$ respectively, and Z_c is the characteristic impedance of the line. Incident voltage and current amplitudes can also be calculated. For all such quantities, the matrices multiplying the source vector do not depend on the line orientation ϕ. These matrices will thus be assembled into a precalculated $N_{tq} \times 2$ product matrix A that multiplies the ϕ-dependent source vector to yield the N_{tq} terminal quantities of interest.

Note that each row of A can assume different line terminations, thus facilitating parameter studies over line terminations in addition to line orientations. Also, some terminal quantities might be related by a single complex factor, enabling further optimization, but we will not discuss that further here.

The time domain terminal quantities are obtained via inverse fourier transform. We will combine pairs of real terminal quantities into complex inverse fourier transforms. Thus only $N_{tq}/2$ complex inverse fourier transforms are required. Furthermore, for n time points we only calculate $n/2 + 1$ frequency points. The remaining points in the fourier transforms are determined by symmetry.

Floating Point Computational Cost

The common numerator

$$n_s = \left(\tilde{E}_{x\theta}\sin\psi\cos\phi - \tilde{E}_{x\phi}\sin\phi\right)\frac{L}{2}$$

can be calculated via two real multiplies of a complex number, and a complex addition. We write the corresponding real operations as 4M, 2A (2FMA), which means either 4M and 2A without any FMAs; or 2M and 2 FMAs.

Each ratio with the common numerator is then one real addition for the denominator, followed by a complex division. Each complex division can be replaced by a complex multiply (4M, 2A (2FMA)), a magnitude squared (2M, A (FMA)), a reciprocal and a real multiplication (D,2M). Including the sum with $\widetilde{V}_t$ makes this last portion D,2M,2A (2FMA).

Given these ratios, the first source term S_1' is then a complex addition followed by a complex multiplication (2A followed by 4M,2A (2FMA)). The second source term S_2' is a complex multiply, a real addition, and another complex multiply (4M,2A (2FMA); 1A; 4M,2A (2FMA)).

The calculations above need $-\chi L$ and $e^{-\chi L}$. The former requires an add (for the integer frequency index) and a multiply. The latter can be calculated via vectorized sin and cos functions, or with nearly equivalent accuracy via sin and cos for a few base χL followed by complex multiplication by a precalculated array of $e^{-\Delta\chi L}$. We will proceed with the latter method and assume the amortized cost of sin and cos for the base χL is negligible.

Each of the N_{tq} terminal quantities then requires two complex multiplications and a complex addition. We will ignore for now that possibility that some of terminal quantities might be related by a single complex factor that would enable some further computational cost savings.

Finally, combining a pair of terminal quantities into a single complex fourier transform requires four real (component) additions per frequency point, accounting for the fact that each of the lower-half frequency points we calculate contributes to two frequency points in the full inverse fourier transform.

The following table summarizes the computational cost per line orientation per frequency point. The last two rows are per terminal quantity N_{tq}, although the actual production subroutine will process terminal quantities in pairs. The total cost per frequency point is $37 + 8N_{tq}$ multiplies (M), $25 + 8N_{tq}$ additions (A), 2 divisions (D), with opportunities for $20 + 6N_{tq}$ fused multiply-adds (FMA). Employing fused multiply-adds means $17 + 2N_{tq}$ multiplies, $5 + 2N_{tq}$ additions, $20 + 6N_{tq}$ fused multiply-adds and 2 divisions.

Table 1. Computational cost by kernel.

result	computation	memory		loop
$-\chi L$	M,A			A
numerator	4M,2A (2FMA)	$4R_M$		A
first ratio/sum	8M,6A (5FMA),D	$4R_M$	$2W_C$	A
second ratio/sum	8M,6A (5FMA),D		$2W_C$	A
$e^{-\chi L}$	4M,2A (2FMA)		$2R_C$	B
first source term	4M,4A (2FMA)	$2R_M$	$2R_C$, $2W_C$	B
second source term	8M,5A (4FMA)		$2R_C$, $2W_C$	B
terminal quantity	8M,6A (6FMA)	$4R_M$	$4R_C$	C
pair combination	2A	$2W_M$		C

The next table presents the theoretical computational cost in clock cycles per double-precision short vector of frequency points, for various processor core architectures, assuming ideal floating-point computational throughput on all floating point vector processing units and instruction ports on a processor core. Each double-precision short vector is 2 (for SSE) or 4 (for AVX) frequency points. The computational throughput of divide units was measured, and is also reported in the table. We furthermore assume divide latency cannot hide any terminal quantity computations beyond the source term: 8M, 8A (6 FMA) per terminal quantity. Division throughput is theoretically a limiting factor on the broadwell processor architecture.

The last two columns present single-thread times for an $n = 2^{23}$ time point calculation with $N_{tq} = 2$ double-precision terminal quantities. Recall we calculate only $n/2 + 1$ frequency points. The inverse

fourier transform (IFFT) times are measured Intel MKL FFT compute times from [2]. If we can obtain 50% of peak performance then calculation of the fourier transforms (the kernels in table 1) will consume only 15% to 25% of the total compute time. Hence the statement in the introduction that we expect the compute time to be dominated by the inverse FFTs.

Table 2. Single-thread theoretical computational cost on various machines/architectures, with $n = 2^{23}$ example.

machine	architecture	division clock cycles per vector	clock cycles per vector	base clock speed (GHz)	$N_{tq} = 2$ time (ms)	IFFT time (ms)
cicero	nehalem	9.276	$39 + 8N_{tq}$	2.67	43.2	253
seneca	ivy bridge	18.732	$39 + 8N_{tq}$	1.70	33.9	241
martial	broadwell	16.400	$32.8 + 5N_{tq}$	2.20	20.4	155
lucretius	coffee lake	4.952	$22 + 5N_{tq}$	2.60	12.9	94

Note that our current BLT implementations are much slower than this estimate. I have one trial c++ code that takes 0.77 seconds per line orientation on cicero, instead of the 0.35 seconds I expect we can achieve. While not ideal, the trial c++ code is 3.8 times faster than the julia implementation.

Memory Transaction Cost

Each row of table 1 also lists real (one floating point number) memory transactions per frequency point: R_M and R_C for a read from main memory or cache respectively; and W_M and W_C for a write to main memory or cache. The reads and writes to cache assume the calculation is tiled and that the stages of the calculation are gathered into three loops (labeled A, B and C) over frequency points. Splitting the tiled calculation into more loops will increase the cache memory traffic, but will not increase the main memory traffic.

The last two rows of table 1 are per terminal quantity, as mentioned earlier. The two writes to main memory ($2W_M$) for the pair combination kernel is per terminal quantity assuming the kernel is processing a pair of terminal quantities. This kernel writes two complex numbers (one for positive frequency, one for negative frequency – four real numbers) per frequency point, but two terminal quantities are being processed, not just one terminal quantity.

An $N_{tq} = 2$ calculation requires $10 + 4N_{tq} = 18$ reads from and $2N_{tq} = 4$ writes to main memory per frequency point, which is $22 \times 8 = 176$ bytes per frequency point assuming streaming stores. The last column in table 3 below lists the main memory bandwidth (BW) required at the theoretical peak floating point computational performance listed in table 2 for $N_{tq} = 2$. A high-performance implementation will consume much of the max available bandwidth to main memory listed in the fourth column, especially on the coffee lake architecture.

The third column lists the minimum number of clock cycles required for loads and stores for one short vector of frequency points. The estimate assumes data are in L1 cache, but we expect some of the data to have been prefetched into L2 cache, which will have a somewhat slower throughput. Nevertheless, for the purpose of this estimate we claim loads and stores are not the limiting factor for theoretical

single-thread performance because fewer clock cycles are required for loads and stores in table 3 than are required for floating point computation in table 2.

Table 3. Theoretical memory costs and rates, with $N_{tq} = 2$.

machine	architecture	load/store clock cycles per vector	main memory max BW per socket (GB/s)	BW required per thread at peak (GB/s)
cicero	nehalem	32	25.6	12.43
seneca	ivy bridge	32	25.6	15.84
martial	broadwell	16	68.3	26.32
lucretius	coffee lake	16	41.8	41.62

Table 3 does make clear that a high-performance implementation running on multiple cores will soon encounter a fundamental main memory bandwidth limitation. Tiling is essential for enabling loads and stores to not be the limiting factor for single-thread performance, but tiling is not enough to let the calculation scale up efficiently to a large number of threads and cores. At 50% of peak floating-point performance, one should expect no more than 5 threads to run effectively on a broadwell socket.

This limitation assumes each thread works on a single line orientation and does not share any data (via cache) with any other thread. Having a thread calculate multiple line orientations for each tile (chunk) of frequency points reduces the thread's demand on main memory bandwidth. The source field, line parameters and terminal quantity coefficients are identical for the extra line orientations, so their values are already in cache. No extra reads from main memory are required, only extra writes. If N_ϕ is the number of line orientations the thread is calculating, then the main memory transaction cost per frequency point is $10 + (4 + 2N_\phi)N_{tq}$ reals, or $144 + 32N_\phi$ bytes for $N_{tq} = 2$.

Table 4. Relative main memory bandwidth requirement, $N_{tq} = 2$.

# of line orientations N_ϕ per thread	# of threads N_{th}			
	1	2	4	8
1	1.000	1.182	1.545	2.273
2	0.591	0.773	1.136	1.864
3	0.455	0.636	1.000	1.727
4	0.386	0.568	0.932	1.659

Table 4 lists the relative main memory bandwidth required per thread as the number of line orientations each thread calculates is increased. The single thread column assumes the thread does not get to share data (via cache) with any other thread.

If multiple threads are sufficiently synchronized to operate on the same tile (chunk) of frequency points while in cache, then the main memory bandwidth requirement increases over the single thread case, but only gradually, as shown in the last three columns of table 4. This suggests nearly all 10 cores of a socket of martial can be employed simultaneously at peak theoretical computational rate, or up to 24 cores of a broadwell socket if the code runs at 50% of peak. The mathematical expression for the

required main memory bandwidth relative to the single thread single line orientation main memory bandwidth BW_{STSL} is

$$\frac{BW_{MTML}}{BW_{STSL}} = \frac{1}{N_\phi}\frac{144 + 32N_\phi N_{th}}{176} \quad \text{for } N_{tq} = 2.$$

Without such synchronization the main memory bandwidth requirement would simply scale with the number of threads, assuming negligible accidental synchronization. Multiply the single thread column in table 4 by the thread count. Without synchronization the prospect for effectively employing a large number of cores quickly fades.

We thus conclude that our fast BLT coupling code should coordinate its threads to operate sufficiently synchronously on the same frequency points (tiles) over multiple line orientations, and allow each thread to likewise calculate multiple line orientations for each tile (chunk) of frequency points. Such synchronization is essential to preventing main memory bandwidth from throttling a threaded (multi-core) calculation of inputs to the inverse FFTs.

Memory Storage Cost

The main memory storage cost for the large arrays is at least $(n/2 + 1)(10 + 4N_{tq}) + n\, N_{tq} N_\phi N_{th} + n\, N_{th}$ doubles, where the last term is scratch space for the inverse FFTs. Storing a second incident field $(\tilde{E}_{x\theta}, \tilde{E}_{x\phi}, \tilde{V}_t)$ changes the first term to $(n/2 + 1)(16 + 4N_{tq})$ doubles, which then enables the coupling calculations to proceed efficiently as one ray (incident field) concludes and another ray starts. For $n = 2^{23}$ and $N_{tq} = 2$, this storage requirement is $0.8053 + 0.1342\, N_{th}(N_\phi + 1/2)$ GB.

Table 5. Main memory storage requirement for $N_{tq} = 2$ and $n = 2^{23}$ time points.

machine	threads N_{th}	lines per thread N_ϕ	bytes per time point	GB required	GB available
cicero	4	15	1088	9.13	24
seneca	2	30	1072	8.99	16
martial	10	6	1136	9.53	192
snow	18	3	1104	9.26	64
lucretius	6	10	1104	9.26	32

Table 5 presents main memory storage requirements for the various machines, assuming $N_{tq} = 2$ terminal quantities are desired for $M_\phi = 61$ line orientations for each ray, hence $N_\phi N_{th} \leq M_\phi$. The storage requirement is mostly dictated by how many line orientations (times terminal quantities) will be stored at once. The fifth column is the storage required for $n = 2^{23}$ time points. The martial and snow rows are per socket. The only machine approaching a main memory storage limitation is seneca.

Table 5 thus shows that main memory storage will not constrain our ability to calculate many line orientations simultaneously for the sake of reducing the demand on main memory bandwidth.

Cache size influences the number of frequency points in each tile (chunk). Each frequency point in the tile uses $16 + 4N_{tq}$ doubles = $128 + 32N_{tq}$ bytes of storage nominally in cache, provided the terminal

quantity fourier transform output is written with streaming stores. For $N_{tq} = 2$ terminal quantities this 196 bytes per frequency point.

A 32 kB L1 cache can thus hold up to 167 frequency points. Ancillary data such as line orientation parameters slightly reduces the number of frequency points that fit in cache. A 256 kB L2 cache can hold up to 1337 frequency points. So for $N_{tq} = 2$ expect each tile to be 150 to 1300 frequency points. This is plenty enough points for vectorized loops to be effective. Loop and tile overhead will be modest.

Not using streaming stores for the terminal quantity fourier transform output causes each frequency point to use an additional 2 doubles of cache, so $16 + 6N_{tq}$ doubles. This would slightly reduce the optimal number of frequency points in a tile.

Data Layout

The array of structures (AoS) of short vectors data layout presented in table 6 facilitates the computation. It reduces the number of index registers required for the computation. It enables the computation to proceed without shuffles, thus reducing instruction count and avoiding the common bottleneck on a core's instruction port 5 [3]. It facilitates hardware and software prefetching.

Four arrays interleave short vectors as shown in table 6. The length in column 2 is the number of frequency components, where n is the number of time points, and m is the number of frequency points in a tile (chunk). Divide by 4 (and round up) for the number of short vector structures, assuming double precision 256-bit AVX instructions and registers.

Table 6. Data layout.

array	length	sv0	sv1	sv2	sv3	sv4	sv5
incident	$n/2+1$	$Re\ \tilde{E}_{x\theta}$	$Im\ \tilde{E}_{x\theta}$	$Re\ \tilde{E}_{x\phi}$	$Im\ \tilde{E}_{x\phi}$	$Re\ \widetilde{V}_t$	$Im\ \widetilde{V}_t$
line	$n/2+1$	$-Re\ \gamma L$	$-Im\ \gamma L$	$Re\ \boldsymbol{e^{-\gamma L}}$	$Im\ \boldsymbol{e^{-\gamma L}}$		
scratch	m	$Re\ S_1'$	$Im\ S_1'$	$Re\ S_2'$	$Im\ S_2'$	$Re\ \boldsymbol{e^{-\Delta\chi L}}$	$Im\ \boldsymbol{e^{-\Delta\chi L}}$
terminal	$n/2+1$	$Re\ A_1$	$Im\ A_1$	$Re\ A_2$	$Im\ A_2$		

The incident field array interleaves 6 short vectors as indicated by columns sv0 through sv5. This incident field array is separate from the line array that interleaves 4 short vectors. Employing separate incident field and line arrays enables the same line array to be used with different incident fields. Separating the line array into separate γ and $e^{-\gamma L}$ arrays would likewise facilitate running calculations with different line lengths, but we do not pursue that option further here.

The scratch array interleaves 6 short vectors ultimately holding the source terms S_1' and S_2', but they also store intermediate results (e.g., the first and second ratios/sums) in the source vector calculation. Reuse of the scratch array in cache is an essential feature of tiling to reduce main memory transactions. The $e^{-\Delta\chi L}$ short vectors are computed prior to the loop over tiles of frequency points, interleaved with the storage for source terms, and likewise kept in cache.

The terminal quantity array interleaves 4 or 8 short vectors, with 4 shown in table 6. Interleaving 8 vectors would store matrix coefficients $A_{11}, A_{12}, A_{21}, A_{22}$ for two terminal quantities. In either case

there can be many terminal quantity arrays, but a kernel will only index one (4 or 8 short vector) or two (4 short vector) terminal quantity arrays at a time.

This data layout means only 3 registers are needed to index the arrays, instead of 8 or 16. Some kernels index the first three arrays in table 6. Other kernels index the scratch and terminal quantity arrays. The trick will be to write code that uses the data layout effectively. Short vector primitive data types and short vector intrinisics are two possibilities, but both of these options are clunky.

The hardware prefetcher only has to follow 2 arrays (incident and line) with this data layout, rather than 5 or 10 arrays if a more traditional data layout were employed. The number of arrays a hardware prefetcher can recognize and follow is usually quite limited. Logic for software prefetch is likewise simplified with just 2 arrays.

Cost Estimate for a Coupling Smile

Consider a sample problem [4] where $N_{tq} = 2$ terminal quantities for $n = 2^{23}$ time points are desired for $M_\phi = 61$ line orientations for $N_r = 697$ rays (ground locations), run on a machine like snow or martial (broadwell xeon processors). The inverse FFTs on martial cost 6590 core seconds. With a parallel speedup of roughly 6 per socket on martial this yields 557 seconds = 9.3 minutes on one node (two sockets). The corresponding inverse FFT compute time on snow is 435 seconds = 7.3 minutes. Note the FFT parallel speedup is limited by main memory latency and bandwidth, not the number of cores.

Computing the inputs to the inverse FFTs as described above takes 1735 core seconds, assuming we obtain 50% of the peak theoretical compute rate. With nearly perfect parallel speedup this becomes 86.7 seconds = 1.45 minutes on martial or 48.2 seconds = 0.80 minutes on snow.

Neglecting terminal quantity data reduction (e.g., peak coupled voltages or currents), the total compute time for the loops over line orientations is 10.8 minutes on martial or 8.1 minutes on one node of snow. Some additional calculation is required for each ray, to calculate the source array. This additional work is mainly an FFT of the two component time profiles of theray's incident EMP. Adding a $2/(M_\phi N_{tq}) = 3.3\%$ fraction to the preceding compute time will cover this per-ray cost. The total compute time is thus 11.1 minutes on martial or 8.3 minutes on one node of snow. This is much faster than the ~10 hours required for the original matlab implementation of the BLT coupling code.

Table 7. Estimated compute times for the sample problem.

machine		compute time (min)
martial	serial	143.3
martial	parallel	11.1
snow node	parallel	8.3

One can presume a good GPU implementation on a very expensive (double-precision capable) GPGPU will be even faster.

Summary

A fast BLT code can calculate coupling smile diagrams of interest in under 15 minutes on a single HPC node or high-performance workstation, much faster than the many hours our current coupling codes require (on the same hardware). The fast BLT coupling code must tile over frequency points, calculate multiple line orientations per tile, and to some degree synchronize tiles over threads, in order to make effective use of individual and multiple cores without running into main memory bandwidth limitations. The fast BLT coupling code should employ computational kernels that eliminate redundant calculation while keeping memory transactions modest, such as the kernels described. An array of structures of short vectors data layout facilitates effective vectorization and prefetching, although the code working with this layout will be clunky.

References

[1] F. M. Tesche, M. Ianoz and T. Karlsson, *EMC Analysis Methods and Computational Models*, Wiley Interscience, December 1996.

[2] Eric M Nelson, "Intel MKL FFT Performance", December 2021.

[3] Intel 64 and IA-32 Architectures Optimization Reference Manual, Intel, June 2016.

[4] Eric M Nelson, "A Strategy for High-Level EMP Vulnerability Assessment", 2 June 2021; LA-CP-21-20508.